ModiChecker: Accessibility Excessiveness Analysis Tool for Java Program

Dotri Quoc　Kazuo Kobori　Norihiro Yoshida

Yoshiki Higo　Katsuro Inoue

In object-oriented programs, access modifiers are used to control the accessibility of fields and methods from other objects. Choosing appropriate access modifiers is one of the key factors for easily maintainable programming. In this paper, we propose a novel analysis method named Accessibility Excessiveness (AE) for each field and method in Java program, which is discrepancy between the access modifier declaration and its real usage. We have developed an AE analyzer - ModiChecker which analyzes each field or method of the input Java programs, and reports the excessiveness. We have applied ModiChecker to various Java programs, including several OSS, and have found that this tool is very useful to detect fields and methods with the excessive access modifiers.

1 Introduction

To realize good encapsulation in Java programs, we have to choose appropriate access modifiers of methods and fields in a class, which may be possibly accessed by other objects. However, inexperienced developers tend to set all of the access modifiers `public` or `none` as `default` indiscriminately.

For example, Figure 1 is a case of bad access modifier setting. Suppose that we have 2 methods: *Method A* and *Method B* in class *X*. *Method A* keeps an initialization process for *Method B*. It means *Method A* must be called before *Method B* is called. Otherwise, *Method B* can not work properly. In this case, *Method B* should be always called via *Method A*, and the access modifier of *Method B* should be set `private`. However, a novice developer might set that access modifier `public` without thinking seriously. In a meanwhile, other developer would want to use *Method B* and he/she can directly call it since the access modifier of *Method B* allows direct access to it. This may cause a fault due to lack of the initialization process performed by *Method A*.

In this example, the access modifier of *Method B* is `public`, but the current program accesses *Method B* from `private` method (*Method A only in this case*) and the access modifier of *Method B* should be `private`. Such discrepancy between the declared accessibility and actual usage of each method and field is called *Accessibility Excessiveness*(*AE*) here.

Existence of AE would be a bad smell of program, and it would indicate various issues on the designs and developments of program as follows.

(1) Immature Design and Programming Issue: An AE would cause unwilling access to a method or

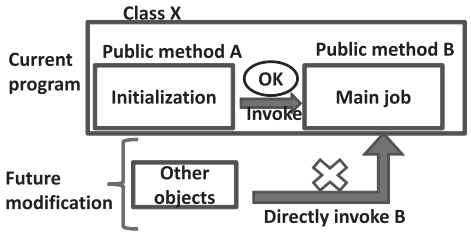

Fig. 1 Inappropriate Access Modifier Declaration

ModiChecker :Java プログラムのアクセス修飾子過剰性分析ツール.

Dotri Quoc, 小堀一雄, 肥後芳樹, 井上克郎, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻, Dept. of Computer Science, Graduate School of Information Science and Technology, Osaka University.

吉田則裕, 奈良先端科学技術大学院大学情報科学研究科, Graduate School of Information Science, Nara Institute of Science and Technology.

コンピュータソフトウェア, Vol.29, No.3 (2012), pp.212–218.

[研究論文 (レター)] 2011 年 10 月 20 日受付.

field which should not be accessed by other objects in a latter development or maintenance phases as shown in the example. This is an issue of design and development processes from the view point of encapsulation [1]. This problem shows the immaturity and carelessness of the designer and developer.

(2) Maintenance Issue: Sometimes developer intentionally set field or method excessive for future use or for the purpose of being called from outsiders. It is not easy to distinguish whether AE is intentionally set by developer or it is a case of Issue 1, so that the maintenance of such program is not straightforward and complicated.

(3) Security Vulnerability Issue: A program with AE has potential vulnerability of its security in the sense that an attacker may access an AE field and/or method against the intention of the program designer and developer [2].

In this paper, we discuss on an AE analysis method mostly focusing on its application to Issue 1 and 2, and Issue 3 will be a further research topic.

We propose an AE analysis tool named *ModiChecker*, which takes a Java program as input, then analyzes and reports the excessiveness of each access modifier declared for each method and field. ModiChecker is based on static program analysis framework *MASU* [6][7][8], which allows a flexible composition of various analysis tools very easily.

Using ModiChecker, we have analyzed several open source software(OSS) such as Ant and jEdit. Also, MASU itself has been analyzed by ModiChecker. The analysis results show that some OSS contain many AE methods and fields, which should be set to more restrictive access modifiers.

There are some previous works related to ours. Tai Cohen studied the distribution of the number of each Java access modifier in some sample methods [3]. Security vulnerability analysis has been studied using static analysis approaches [4]. Among these researches, an issue of access modifier declaration has been discussed by Viega et al. [2], where a prototype system Jslint has been presented without any detailed explanation of its internal algorithm and architecture. Also, Jslint only gives warning for the fields/methods which are undeclared private, while our tool supports all kinds of access modifier declarations based on analyzing actual usage.

In the following, we will define AE in Section 2.

Table 1 Accessibility Excessiveness Map

Declaration \ Actual Usage	Public	Protected	Default	Private
Public	ok-pub0	pub1	pub2	pub3
Protected	x	ok-pro0	pro1	pro2
Default	x	x	ok-def0	def1
Private	x	x	x	ok-pri0

Section 3 describes ModiChecker and MASU. In Section 4, we will show our experimental results. Section 5 will conclude our discussions with a few future works.

2 Accessibility Excessiveness Map

Table 1 is called *Accessibility Excessiveness Map* (AE map), which lists all the cases where an AE happens. The vertical column shows the declaration of an access modifier for a method or field in the source code. The horizontal row shows its actual usage from other objects. Each element in AE map is an *AE Identifier* (*AE id*) which identifies each AE case. For example, if a method has `public` as the declaration of the access modifier, and it is accessed only by the objects of same class, the AE id is "pub3" meaning it could be set to `private`. Note that "`default`" means the case that there is no explicit declaration of the access modifier and it is the same as `package`.

An AE id "`ok-xxx`" means that there is no discrepancy between the declaration and actual usage, and it is an ideal way of quality programming. An AE id "`x`" means that these cases are detected as error at the compilation time and they are out of the scope of the AE analysis. An AE id in shaded cells means that the declaration is excessive one from the actual usage of the access modifier.

Purpose of the AE analysis is to identify an AE id for each method and field in the input source code. Also, we are interested in the statistic measures of AE ids for the input program, which would be important clues of program quality.

3 AE Analysis Tool ModiChecker

3.1 Approach to AE Analysis

To perform the AE analysis, we need to know the declaration of the access modifiers of each method and field of the input program. This is easily done by parsing the program. Also, we have to investigate into the actual usage of each method and field.

For this work, we employ a static source-code analysis, which identifies other classes that may possibly access the target method or field. For these purposes, we have used a Java program analysis framework MASU [6][7][8].

MASU has been originally designed to implement pluggable multi-purpose metrics infrastructure, but it is very useful as a Java program analysis framework. MASU transforms the input Java program into an Abstract Syntax Tree (AST), and then it analyzes AST for actual usage of the methods and fields in the input.

ModiChecker is a system with 521 source files and 102,250 LOC in Java, developed based on MASU framework [8]. MASU framework itself accounts for 519 source files and 102,000 LOC in Java (41,000 LOC are automatically generated code by ANTLR [9]).

3.2 Overview of ModiChecker Architecture

Figure 2 shows the architecture of ModiChecker. Firstly, ModiChecker reads source program and all of the required library files (normally, the library files are often in .jar files) in Java. The source code is transformed to an AST associated with various static code analysis results.

After analyzing source, we get the access modifier declaration and also usage of each field and method. From the AST database, we can easily know which class may access that method/field.

By comparing the declaration of the access modifier and real usage of the field and method, ModiChecker reports AE for each field and method.

ModiChecker treats some special cases as follow:

- ModiChecker does not give any report for methods of abstract classes or interfaces because they are overridden by the method of other classes. One more reason is that an abstract class or interface does not generate any object so that its methods will never be called and those access modifiers do not affect maintenance processes.
- In the case of a method overriding another method, the overriding method in a subclass must have an access modifier with an equal or more permissive level to the access modifier of the overridden method. ModiChecker detects such an overriding method and reports an AE id between the access modifier of the overridden method and its actual usage. For example, in Figure 3, assume that we have two classes *Class A* and *Class B* with *Method A.C* and *Method B.C* of access modifier `public` for both. *Method B.C* overrides *Method A.C* so ModiChecker does not report `private` for *Method B.C* even if *Method B.C* is actually used inside *Class B* only. In dynamic binding cases, if a class accesses a method of a superclass, ModiChecker will consider that class also accesses the method of all subclasses of the superclass. By this way, dynamic binding cases should not bring about any bad effect to ModiChecker analysis result.

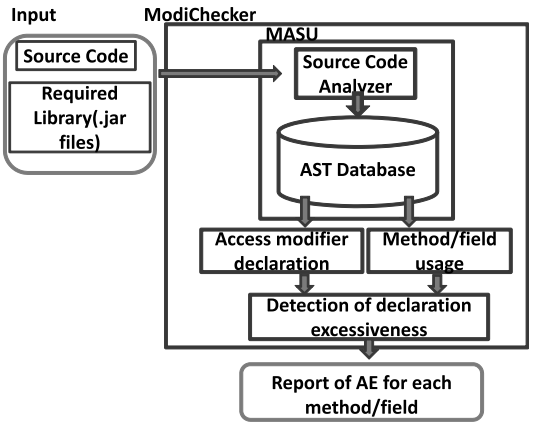

Fig. 2 Architecture of ModiChecker

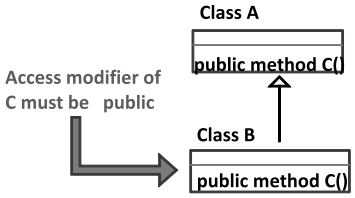

Fig. 3 Access Modifier of Overriding Method

4 Experiments and Discussions

4.1 Overview

We have conducted case studies with some open-source code projects to evaluate the AE analysis. In the evaluation, we have focused on the following points.

- The total number of each AE id is measured to evaluate how program is well designed.
- Based on the above result, we have closely

Table 2 Size and Running Time of Target

Program	Size	Running Time (mins)
MASU	519 files/102,000 LOC	1.81
ANT	1,141 files/127,235 LOC	5.65
jEdit	546 files/109,479 LOC	6.56

Table 3 Number of Detected AE ids for Fields in MASU

Declaration \ Actual Usage	Public	Protected	Default	Private
Public	16(6.4%)	0	3(0.4%)	259(33.0%)
Protected	x	0	1(0.1%)	14(1.8%)
Default	x	x	0	3(0.4%)
Private	x	x	x	488(62.2%)

Total number of fields: 784
Total number of fields in shaded cells: 280(35.7%)

Table 4 Number of Detected AE ids for Methods in MASU

Declaration \ Actual Usage	Public	Protected	Default	Private
Public	471(26.9%)	3(0.2%)	90(5.1%)	124(7.1%)
Protected	x	19(1.1%)	0	35(2.0%)
Default	x	x	4(0.2%)	1(0.1%)
Private	x	x	x	1006(57.4%)

Total number of methods: 1753
Total methods in shaded cells: 253(14.43%)

investigated the reasons of setting the access modifiers excessively. Sometime an access modifier would be intentionally set excessively by the developers for the future purpose, or sometime they would be set excessively by automatic code generator.

The target software products are 3 Java programs: MASU, Ant 1.8.2, jEdit 4.4.1.

We did these experiments on a PC workstation with the specification of OS : Windows 7 Enterprise 64bit, CPU : Intel Xeon 5160(3.00 GHZ, 2 processors), Memory: 8.0 GB.

Table 2 shows the size and running time of each experiment.

4.2 Experiment Result

4.2.1 MASU

By analyzing MASU, we got the number of detected AE ids as shown in Table 3 and Table 4.

We have found 280 fields with the excessive access modifiers. Out of these 280 fields, 255 fields were identified as automatically generated code by our hand-analysis. We have interviewed the developer of MASU and asked the reason of the excessiveness of other fields. 20 fields have been found that they

Table 5 Number of Detected AE ids for Fields in Ant

Declaration \ Actual Usage	Public	Protected	Default	Private
Public	157(4.9%)	22(0.7%)	64(2.0%)	285(8.9%)
Protected	x	54 (1.7%)	47(1.5%)	135(4.2%)
Default	x	x	21 (0.7%)	58(1.8%)
Private	x	x	x	2395(73.8%)

Total number of fields: 3238
Total number of fields in shaded cells: 611(18.9%)

Table 6 Number of Detected AE ids for Methods in Ant

Declaration \ Actual Usage	Public	Protected	Default	Private
Public	1576(36.8%)	100(2.3%)	609(14.2)	454(10.6%)
Protected	x	103(2.4%)	117(2.7%)	217(5.1%)
Default	x	x	52(1.2%)	23(0.5%)
Private	x	x	x	1034(24.1%)

Total number of methods: 4284
Total number of methods in shaded cells: 1519(35.5%)

are intentionally set excessively for future uses. Finally, 5 fields have been identified actually excessive and those access modifiers have been changed to proper ones.

We have also found 253 methods with the excessive access modifier. And by our hand-analysis, 6 methods were found to be automatically generated code. Out of those 253 methods, 181 methods are intentionally set excessively for future uses. Finally, 66 methods have been identified actually excessive and those access modifiers have been changed to proper ones.

4.2.2 Ant 1.8.2

We have investigated into the newest version of Ant 1.8.2 and got the number of detected AE ids for fields and methods as shown in Table 5 and Table 6.

We have found 611 fields and 1520 methods with the excessive access modifiers. By our hand-analysis, we were unable to find any field with excessive access modifier generated by some automatic code generator.

Looking at the ratio of excessive access modifier, the ratio of excessive fields is 18.9%(shown in the shaded cells in Table 5) while ratio of excessive methods is 35.5%(shown in shaded cells in Table 6). Since a standard design strategy might be to make all fields private and to provide public getter/setter methods for them, methods has more probability to be set excessively for future use than fields. That would be the reason why the ratio of excessive methods is higher than ratio of excessive fields.

Table 7 Number of Detected AE ids for Fields in jEdit

Declaration \ Actual Usage	Public	Protected	Default	Private
Public	228(9.1%)	10(0.4%)	111(4.4%)	163(6.5%)
Protected	x	23(0.9%)	15(0.6%)	51(2.0%)
Default	x	x	126(5.0%)	254(10.1%)
Private	x	x	x	1529(60.9%)

Total number of fields: 2510
Total number of fields in shaded cells: 604(24.1%)

Table 8 Number of Detected AE ids for Methods in jEdit

Declaration \ Actual Usage	Public	Protected	Default	Private
Public	1224(34.5%)	77(2.4%)	544(16.7)	237(7.3%)
Protected	x	44(1.4%)	14(0.4%)	23(0.7%)
Default	x	x	233(7.2%)	86(2.6%)
Private	x	x	x	874(26.8%)

Total number of methods: 3223
Total number of methods in shaded cells: 981(30.4%)

4.2.3 jEdit 4.4.1

The result of detected AE ids for fields and methods for jEdit 4.4.1 is shown in Table 7 and Table 8.

For jEdit, we have found 604 fields and 981 methods with the excessive access modifiers. We were unable to find any field or method with excessive access modifier generated by some automatic code generator by our hand-analysis.

For jEdit 4.4.1, the ratio of excessive fields is 24.5%(shown in the shaded cells in Table 7) while the ratio of excessive methods is 30.4%(shown in the shaded cells in Table 8). Like the case of Ant 1.8.2 shown above, the ratio of excessive fields is lower than the ratio of excessive methods.

4.3 Discussions

To validate the analysis result of ModiChecker, we have changed all the excessive access modifiers of above three programs to suggested access modifiers. All the modified programs have been compiled and executed without any error. This indicates that the output report of ModiChecker is proper one in the sense that the reported excessive access modifiers can be changed to more restrictive access modifiers without causing any error.

As mentioned before, our tool gives the developer the AE analysis result for each field/method in the current target program, but it still can not make sure that some of them were intentionally set excessive for future use. Only designer and developer can identify that those fields/methods are really excessive or not. Thus, we need a tool by which a developer can select each access modifier found as excessive and to be changed to more restrictive one by her/his decision.

By using AE analysis, we could propose quality metrics in the following ways.

- We set a value called AE index for each AE id and sum up each AE index as metrics value.
- We set a value for each method and field based on the number of other classes accessing those fields and methods. Those values for each method/field are accumulated as this metrics value.

These metrics would indicate bad smell of program such as immaturity of design and implementation, low maintainability, security vulnerability, and so on. We need further experiments for the validation of effectiveness of such metrics.

The idea of using access modifier metrics would be related to our previous work [5]. In that paper, the number of each Java access modifier is used as one of the metrics for checking the similarity between Java source codes.

To recognize intentional AE fields/methods for future use without interviewing developers is not easy. As a simple estimation method, we propose the following approach which can figure out some part of the excessive fields/methods for future use, using test files associated with the target program.

The first step is checking the source files without test files and we get the result of ModiChecker for the target program itself. Then we add the test files and check them together without counting the fields/methods in test files. The excessive fields/methods found in the first step but not in the second step could be the fields/methods for future use, since they were actually accessed by the objects of test files. The test designer anticipated that those fields/methods should be access from outside the program in the future.

In this experiment we have not counted for the fields/methods which are not accessed by any objects (we would like to call them no access fields/methods). The reason of no access fields/methods might be developers' carelessness and intentional future use. Investigating into no access fields/methods would be an interesting future research topic.

5 Conclusions

In this paper, we have proposed an analysis method named AE for each field and method in Java program, which is discrepancy between an access modifier declaration and the real usage of the field and method. We have also introduced AE Map which lists all of the cases where an AE happens.

We have developed a tool named ModiChecker, which finds excessive method/field and reports AE id of each excessive method/field. We have also used ModiChecker to analyzed several OSS such as MASU, Ant, jEdit, and found that our system is quite useful to detect fields and methods with the excessive access modifiers.

Since there is no other tool to analyze access modifiers as discussed here, we think ModiChecker will be an important tool to support quality programming in Java.

Currently we are analyzing other Java programs including industrial systems, and are trying to identify the relation between the AE analysis results and other program quality indicators such as bug frequency.

Acknowledgements This work is supported by JSPS, Grant-in-Aid for Scientific Research (A) (No. 21240002) and Grant-in-Aid for Exploratory Research (No. 23650015). This is also supported by MEXT Stage Project, the Development of Next Generation IT Infrastructure.

References

[1] Booch, G., Maksimchuk, R.A., Engel, M.W., Young, B.J., Conallen, J. and Houston, K.A.: *Object-Oriented Analysis and Design with Applications*, Addision Wesly, 2007.

[2] Viega, J., McGraw, G., Mutdosch, T. and Felten, E.: Statically Scanning Java Code: Finding Security Vulnerabilities, *IEEE software*, Vol. 17 No. 5(2000) pp. 68–74.

[3] Cohen, T.: Self-Calibration of Metrics of Java Methods towards the Discovery of the Common Programming Practice, *The Senate of the Technion*, Israel Institute of Technology, Kislev 5762, Haifa, 2001.

[4] Evans, D. and Larochells, D.: Improving Security Using Extensible Lightweight Static Analysis, *IEEE software*, Vol. 19, No. 1(2002), pp. 42–51.

[5] Kobori, K., Yamamoto, T., Matsushita, M. and Inoue, K.: Java Program Similarity Measurement Method Using Token Structure and Execution Control Structure, *Transactions of IEICE*, Vol. J90-D No. 4 (2007), pp. 1158–1160.

[6] Higo, Y., Saito, A., Yamada, G., Miyake, T., Kusumoto, S. and Inoue, K.: A Pluggable Tool for Measuring Software Metrics from Source Code, in *The Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, 2011 (to appear).

[7] Saito, A., Yamada, G., Miyake, T., Higo, Y., Kusumoto, S. and Inoue, K.: Development of Plug-in Platform for Metrics Measurement, in *International Symposium on Empirical Software Engineering and Measurement*, Poster Presentation, Lake Buena Vista, 2009.

[8] MASU, http://sourceforge.net/projects/masu/

[9] ANTLR, http://antlr.org

[10] Ant, http://ant.apache.org

[11] jEdit, http://jedit.org

Dotri Quoc

2005 年ベトナムの郵政電信工芸学院 (Institute of Posts and Telecommunications Technology) に入学，2007 年同大学中退．2008 年大阪大学基礎工学部情報科学科入学，2012 年卒業．現在楽天 (株) に所属．日本ソフトウェア科学会会員．

小堀一雄

1979 年生．2005 年大阪大学大学院情報科学研究科修士課程修了．同年 (株)NTT データ入社．ソフトウェアテストやソフトウェアプロセスの研究に従事．2010 年大阪大学大学院情報科学研究科博士課程入学．

吉田則裕

2004 年九州工業大学情報工学部知能情報工学科卒業．2009 年大阪大学大学院情報科学研究科博士後期課程修了．2010 年奈良先端科学技術大学院大学情報科学研究科助教．博士 (情報科学)．コードクローン分析手法やリファクタリング支援手法に関する研究に従事．ソフトウェア科学会，情報処理学会，

電子情報通信学会，人工知能学会，IEEE，ACM 各会員．

肥後芳樹

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程修了．2007 年同大学大学院情報科学研究科コンピュータサイエンス専攻助教．博士 (情報科学)．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．日本ソフトウェア科学会，情報処理学会，電子情報通信学会，IEEE 各会員．

井上克郎

1984 年大阪大学大学院基礎工学研究科博士後期課程修了 (工学博士)．同年，大阪大学基礎工学部情報工学科助手．1984〜1986 年，ハワイ大学マノア校コンピュータサイエンス学科助教授．1991 年大阪大学基礎工学部助教授．1995 年同学部教授．2002 年大阪大学大学院情報科学研究科教授．2011 年 8 月より同研究科研究科長．ソフトウェア工学，特にコードクローンやコード検索などのプログラム分析や再利用技術の研究に従事．